VŠB – TECHNICAL UNIVERSITY OF OSTRAVA

FACULTY OF ELECTRICAL ENGINEERING

AND COMPUTER SCIENCE

DIPLOMA THESIS

DICTIONARY BASED DATA COMPRESSION

2017 Khalifa Saghair

VŠB – TECHNICAL UNIVERSITY OF OSTRAVA

FACULTY OF ELECTRICAL ENGINEERING

AND COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

DICTIONARY BASED DATA COMPRESSION

SLOVNÍKOVÁ KOMPRESE DAT

2017 KHALIFA SAGHAIR

VŠB - Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Diploma Thesis Assignment

Student: **Khalifa Ahmed Saghair**

Study Programme: N2647 Information and Communication Technology

Study Branch: 2612T025 Computer Science and Technology

Title: Dictionary Based Data Compression
Slovníková komprese dat

The thesis language: English

Description:

The aim of this work is to map the current situation in the field of dictionary based compression algorithms. Main focus should be on the LZRW algorithms and its variants.

The work should contain the following parts:
1. State of the Art – a literature review for various dictionary based techniques.
2. Description of the selected algorithm.
3. Implementation of the algorithm.
4. Experimental evaluation and comparison with other algorithms.

References:

[1] Handbook of Data Compression, Salomon D., Motta G., Springer; 5th edition, 2010
[2] An Introduction to Statistical Signal Processing, Gray R.M., Davisson Lee D., 2004, Cambridge University Press
[3] Data Compression: The Complete Reference, Salomon D., Springer; 4th edition (December 19, 2006), ISBN: 978-1846286025
[4] Variable-length Codes for Data Compression, David Salomon, Springer; 1st Edition. edition (October 1, 2007), ISBN: 978-1846289583

Extent and terms of a thesis are specified in directions for its elaboration that are opened to the public on the web sites of the faculty.

Supervisor: **doc. Ing. Jan Platoš, Ph.D.**

Date of issue: 01.09.2015

Date of submission: 28.04.2017

doc. Dr. Ing. Eduard Sojka
Head of Department

prof. RNDr. Václav Snášel, CSc.
Dean

I hereby declare that this master's thesis was written by myself. I have quoted all the references I have drawn upon.

Ostrava, July 18, 2017 …….

Abstract

Data compression has gained increased attention of researchers and developers in the last few decades. It comes handy when users have limited storage capacity or transmission bandwidth as it reduces the size of data without much loss. There have been many types of compression techniques proposed by researchers so far, Lempel-Ziv series of algorithms, for example. Lempel-Ziv series of algorithms are lossless algorithms and follow a dictionary-based approach to data compression where a dictionary is used to keep references of repeated text or words and are omitted to reduce size of data. Out of all the Lempel-Ziv algorithms, this study focuses on Lempel-Ziv-Ross-Williams (LZRW) algorithm and its implementation, which was proposed by Ross Williams in 1991. There is a brief introduction in the study which talks about basics of data compression followed by the compression techniques. Various dictionary-based algorithms have also been compared with each other discussing along with their weaknesses and advantages. Not only that, there is also a brief description on how data compression works for L1 cache. The design issues faced by the designers while implementing data cache compression have also been noted in this study.

Keywords*: Data Compression; Lempel-Ziv algorithm; Lempel-Ziv-Ross-Williams; LZRW; Data Cache Compression.*

Abstrakt

Komprese dat si získala v posledních několika desetiletích zvýšenou pozornost výzkumníků a vývojářů. Je užitečné používání komprese dat, pokud uživatelé mají k dispozici omezenou úložnou kapacitu nebo datovou propustnost, protože snižují velikost dat bez velké ztráty. Existuje mnoho typů kompresních technik navržených výzkumnými pracovníky, např. řady algoritmů Lempel-Ziv. Série algoritmů Lempel-Ziv jsou bezztrátové algoritmy a řídí se slovníkem založeným na kompresi dat, kde je slovník používán k udržování odkazů na opakovaný text nebo slova a je vynechán ke snížení velikosti dat. Ze všech algoritmů Lempel-Ziv se tato studie zaměřuje na algoritmus Lempel-Ziv-Ross-Williams (LZRW) a jeho implementaci, který navrhl Ross Williams v roce 1991. Ve studii je krátký úvod, který hovoří o základech komprese dat s popisem kompresních technik. Různé algoritmy na bázi slovníku byly také navzájem porovnány, spolu s jejich výhodami a nevýhodami. Studie obsahuje také stručný popis toho, jak komprese dat funguje pro vyrovnávací cache paměť L1. V této studii byly rovněž zaznamenány konstrukční problémy, s nimiž se potýkají vývojáři při implementaci komprese dat ve vyrovnávací paměti.

Klíčová slova: komprese dat; algoritmus Lempel-Ziv; Lempel-Ziv-Ross-Williams; LZRW; komprese dat ve vyrovnávací paměti.

ACKNOWLEDGEMENT

This thesis forms the final part of my Master's program; at VŠB - Technical University of Ostrava.

I would like to thank the people who have helped me in several ways with this project. First, I want to express my deep thanks to my supervisor doc. Ing. Jan Platoš, Ph.D. for their professional guidance and support during this project. Secondly, I would like to thank my friends Ing. Hussam Abdullah Ph.D. and Mr. Fathi Ali for their help and support to me during my stay in the Czech Republic.

Lastly, I would like to thank my wife and son for their patience and support. I am wishing to see my son can stand and walk soon. Without those people, this thesis would not be as successful as it is now. Working with so many people on such an exciting subject in a national environment has been a lifetime experience for me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1: INTRODUCTION

1.1 Background of the Research

Data compression is all about reducing statistical redundancy in data, where Redundancy refers that part of data which can be removed without losing any essential information (Blelloch, 1998). The process of compression is used to reduce the size of input data by generating a duplicate which uses fewer bits. Data size is reduced to conserve resources used in storing or transmitting the data. This technique of data compression has been an important subject for researchers in last few decades as there are many algorithms proposed to implement data compression in various ways such as the Lempel-Ziv algorithms (LZ77, LZW, LZRW), Huffman coding, DEFLATE and many others. Some algorithms work by using a dictionary while some depend on coding for most of the work (Lelewer & Hirschberg, 1987; Nelson & Gailly, 1995). The dictionary-based algorithms use a dictionary to keep references of part of the text and using those references whenever a repeated text is entered. This results in reduced size of the data as the references can be used to point to the original data during decompression (Salomon & Motta, 2010).

Various algorithms have been developed over the last few decades using the dictionary-based technique such as the Lempel-Ziv (LZRW, LZW, LZ77) and other similar algorithms. Algorithms can be of two types– lossless compression and lossy compression. In the lossless algorithms, the exact source file is generated after the decompression that is why it is known as lossless data compression. On the other hand, in lossy data compression the algorithms remove the non-essential data from the input file to reduce the size such that the same original file cannot be generated after decompression (Blelloch, 1998). The above mentioned LZRW, LZW and LZ77 are lossless data compression algorithms. These dictionary-based data compression algorithms generate a dictionary or index to replace the repeated occurrences from the original file and reduce the bits. All the above mentioned dictionary-based algorithms follow a similar approach to create the dictionary, but they differ by their way of implementation, and the method used for compression and decompression (Mahmud, 2012). This research will focus on LZRW and its variants which were developed in 1990s by Ross Williams. Ross Williams did years of research on the series of Lempel-Ziv algorithms already proposed and came up with his own data compression algorithm to tackle the weaknesses of other Lempel-Ziv algorithms.

All the Lempel-Ziv algorithms are named after the researchers who contribute in developing the algorithm. For instance, LZRW stands for Lempel-Ziv-Ross-Williams, is a dictionary based data compression algorithm and a member of LZ77 class algorithms. LZRW1 uses the LZ77's approach of single pass literal/copy mechanism and is based on the A1 algorithm developed by Fiala and Greene. LZRW was created by Ross Williams in 1990s with fast execution as the primary objective.

The first version, LZRW1 was developed in 1991 whereas the other variants were created afterwards. After LZRW1, the variants released by Ross Williams were LZRW1-A, LZRW2, LZRW3, LZRW3-A, LZRW4 and LZRW5. In LZRW1, compression is achieved by dividing the input text into literal items and copy items, representing the input in copy items as much as possible and using the literal items only when a match cannot be found in the dictionary. When the input is broken into literal or copy items, literal items contain the text they represent while the copy items contain the offset and the point to the substring already transmitted (Williams, 1991).

1.2 Problem Statement

Several dictionary-based lossless compression algorithms have been proposed so far, such as the variants of Lempel-Ziv algorithms. Each of them has its own weaknesses and advantages in their usage of encoding and decoding, thus affecting the overall performance. This study compares those data compression algorithms by their weaknesses and the differences in approach used in their implementation. In their application of data cache compression, the study also talks about the various issues which the designers face while designing the cache memory and the solutions to those design issues.

1.3 Research Aim & Objectives

The research aims at providing a better understanding of Lempel-Ziv-Ross-Williams dictionary-based data compression algorithm by studying its approach, the procedure it follows, and its original implementation in depth along with, comparing it with other available dictionary-based algorithms. The objectives of this research are as follows:

- To study LZRW data compression algorithm and its procedure in depth.
- To provide a comparison of LZRW with other available dictionary-based data compression algorithms.
- To study the design issues faced by developers in data compression.
- To implement the algorithm in a data compression program.

1.4 Research Questions

As mentioned above, various algorithms have been developed by researchers which aim at reducing size of various types of data files, such as images, text, audio and video whether it is about storing the data or transmitting to somewhere else. But few questions come to mind.

RQ1: There are a number of Lempel-Ziv algorithms. How do they differ from each other?

RQ2: How do those algorithms actually work? What is their implementation process?

RQ3: In what way LZRW is better/lagging behind other data compression algorithms?

1.5 Significance of the study

Over the last few decades, majority of data compression software have been developed based on Lempel-Ziv series of algorithms, such as GZip, LHarc and others. They have also been used in various compressed image formats such as GIF and JPEG. The Unix compression utility 'compress' was also made on the basis of a Lempel-Ziv algorithm- LZW- when it gained popularity. This study provides a basic introduction of the major Lempel-Ziv algorithms along with their disadvantages. Furthermore, the study shows how the Lempel-Ziv-Ross-Williams is implemented by explaining the procedure involved and making a JAVA-based program on it.

Chapter 2: LITERATURE REVIEW

2.1 Introduction to the Chapter

As discussed in the previous chapter, data compression is a process of transforming or encoding data into a compact form which uses fewer bits than its original form. The compressed data takes less space. This phenomenon is useful where the disk capacity is a limiting factor or where the data needs to be transmitted faster than the transmission rate supported by the network (Nelson & Gailly, 1995). In case of transmitting the data, it can be compressed at one end/sender and decompressed at the other end/receiver. Thus, data compression reduces the resources used to store and transmit data. The process of compressing the data for the purpose of transmission is termed as source coding. The encoding is done at the sender's end and decoding takes place at the receiver's end. The process of encoding and decoding takes place with the help of a key. Using a key for encoding and decoding makes sure that the data does not go in wrong hands. The process of data compression is all about redundancy. Redundancy is the part of message which can be removed from the input data without losing any valuable information. Thus, the whole process of data compression revolves around redundancy (Zhang, Yang, & Gupta, 2000).

As discussed in the preceding chapter, there are two compression techniques based on the nature of user's application. If the user wants best compression and can accept loss of few bits, which involves removing non-relevant information, then the lossy compression provides the best compression ratio. While for those uses where the data needs to resemble exactly the original data without any loss, the lossless compression is used. The lossless compression algorithms usually provide low compression ratios than lossy compression algorithms. Even the users who do not have much knowledge about compression must have heard about GIF and JPEG image formats. Those image formats work on compression algorithms where the source image is compressed to take lesser disk space. The JPEG format works on lossy compression techniques while the GIF image format works on lossless compression algorithms such as the Lempel-Ziv class of lossless algorithms.

The current chapter covers the various dictionary-based data compression techniques such as LZ77, LZ78 and others while comparing their advantages and disadvantages. The data compression can also be seen in the cache memory of the machines people use, as has been explored below. Later, a brief introduction of data cache compression and the design issues faced in designing it are also mentioned, concluding the chapter with details of the selected algorithm.

2.2 Dynamic behavior of frequent values

Data compression has found application in data cache compression where the data entering into the cache memory is compressed at the cost of latency. L1 and L2 cache have always been limited for

the systems which has made cache compression an important subject for researchers and an objective for hardware developers. Cache compression reduces the memory taken by data inside the cache, thus, resulting in increased apparent data capacity of the cache memory. Increase in data capacity, in turn, results in reduced miss-rate but it comes at the cost of increased access latency due to the frequent compression and decompression (Alameldeen & Wood, 2004). The use of compression techniques depends on the applications. The compression turns out to be useful for the user only if the cost of accessing compressed data in the cache does not exceed the cost of servicing a miss from the lower level of the memory hierarchy. L1 cache has always been very small in size, usually in few kilobytes. There have been several techniques introduced for L2 compression such as adaptive cache compression, but compression for L1 cache still remains a goal due to its small size and high sensitivity to latency. The cache compression is easily achievable in case of L2 and L3 caches where the loss caused due to compression and decompression expenses is certainly lower as compared to accessing the main memory. Therefore, implementing data compression is even easier, and there has been varieties of data compression algorithms for main memory (Keramidas, Aisopos, & Kaxiras, 2006).

In 2006, Georgios Keramidas, Konstantinos Aisopos and Stefanos Kaxiras came up with the first dynamic dictionary-based compression technique for L1 cache which surpassed the previously static dictionary-based techniques by good margins in terms of power, hit ratio and energy delay product (Keramidas et al., 2006). It was based on the phenomenon 'Frequent Value Locality' introduced by Youtao Zhang, Jun Yang and Rajiv Gupta in the year 2000. The frequent value locality phenomenon works on the frequently occurring values by characterizing the behavior of values being held in live memory locations of running programs (Zhang et al., 2000). Those frequently occurring values were termed "frequent values". The technique was limited by static dictionaries which accommodated only a small number of frequent values. This resulted in limited execution of the compression technique. The technique, based on dynamic dictionaries proposed by Keramidas et al., (2006) overcame the limitations created by static nature of dictionaries. The contents of a dynamic dictionary change dynamically with the potential of better performance and lower power consumption. Implementation of dynamic dictionaries also frees up the designers from initializing it properly. Although there have not been much applications of dynamic dictionaries in cache compression, but it has seen some light in case of bus compression where the dynamic dictionaries do not require to be kept consistent with any other state and the data is compressed instantaneously as they enter the bus and decompressed right when they are delivered to the other end.

The frequent value technique proposed also takes care of the particular issue, where the compressed cache data needs to be kept consistent at all times with the dictionary contents, by using cache decay.

Cache decay identifies the cache lines which are supposed to be less frequently or not at all accessed in the future. Thus, those cache lines can be turned off to save energy resulting in lower power consumption. This technique uses decay for both cache and the dictionary in exactly the same way ensuring that whenever a dictionary entry is decayed, it can no longer be referred by any live line in the cache (J. Yang, Zhang, & Gupta, 2002). Decay, here, is implemented by measuring the time since last access was made to a cache line/dictionary entry. If a specified time interval passes without any access, then the cache line or dictionary entry is discarded. That specified time interval is known as decay interval. The decay interval is measured by using counters in each cache line or dictionary entry. The counters advance when the cache line or dictionary entry is idle and are reset whenever they are accessed. When a counter reaches the decay interval, its corresponding cache line or dictionary entry is decayed or marked empty. That is how the decay identifies which cache line needs to be turned off to save power. Only the cache lines are turned off as in the case of dictionary entries, they are simply marked as empty so that they cannot be referred to in future. The decay intervals usually range in a few thousand cycles. Thus, this implementation uses a hierarchical counter scheme where a global cycle counter advances every few hundred cycle's small local counters in each line/entry, to show which, the decay keeps the compressed cache and the decaying dictionary consistent at all times (Yang et al., 2002).

Frequent values of a live line are kept in the dictionary for at least a decay interval. As a result, when a frequent value decays in the dictionary, it means that no cache line that uses that particular frequent value for its compression has been accessed for a full decay interval otherwise the frequent value would not have been decayed and would still be live. This condition also signifies that all of the cache lines compressed with this frequent line have also decayed which gives the possibility to replace the decayed entries in the dictionary with new frequent values. This process makes the contents of the dictionary suitable to the set of frequent values which are most useful during various phases of the execution. The entries are not replaced on demand, but are rather replaced according to the availability of decayed entries (Kaxiras & Martonosi, 2008).

L1 cache compression is not that easy to implement and requires proper expertise to design it in a very cautious manner. There are few design issues that designers face while implementing Zhnag's Dynamic Frequent Value Cache (DFVC) technique. The issues occur while designing decaying dictionaries, Power-Aware DFVC (PA-DFVC) and High-Performance DFVC (HP-DFVC) (Keramidas et al., 2006).

2.3 Types of compression

Data compression that are widely used are essentially of two types– lossy compression and lossless compression. In the case of Lossy data compression, some loss of bits is acceptable, hence the non-

essential data can be removed to achieve a better compression ratio. Lossy data compression consists of those algorithms which aim on achieving a high compression ratio by deleting the irrelevant data from the input stream. The data compressed using lossy algorithms cannot be reconstructed back exactly to the original file which is usually the only trade-off with lossy data compression techniques. Thus, these algorithms are used where a slight loss of data can be accepted such as while transmitting a speech. If the quality of the transmitted speech is similar, then it can be accepted. Same goes with the video transmission, slight degradation in quality is usually accepted. JPEG (Joint Photographic Experts Group) is quite common for its use in image compression. It works by removing the less important details from the image and reducing the size of original BMP (Bitmap) image files drastically (Sayood, 2012).

Lossless data compression on the other hand, uses those data compression algorithms and techniques which generate the exact same original file after a compression-decompression cycle. There is no loss of information accepted in this case which puts a limit on how much compression can be achieved, unlike lossy compression. Lossless compression techniques and algorithms are generally used in storing database records, spreadsheets, documents files where the loss of even a single bit would create a problem. Statistical redundancy makes lossless data compression possible. The Lempel-Ziv compression algorithms are popular for their lossless compression techniques. Graphical Interchange File (GIF), commonly used over the web, is a good example of lossless image compression which uses LZW (Lempel–Ziv–Welch) compression algorithm. LZW is an improved version of LZ78 which was developed by Lempel and Ziv in 1984. While the Zip methods are based on LZR (LZ-Renau) methods. DEFLATE is yet another example of lossless data compression algorithm which uses LZ77 and Huffman coding for better decompression speed and compression ratio. All the LZ algorithms follow the same dictionary based approach where an index is created and all the repeated entries in the input are replaced with the index entries. Modern data compression techniques use probabilistic models for data compression, such as prediction by partial matching (Mahmud, 2012).

2.4 Dictionary Based Data compression techniques

Certain applications consist of source output based on recurring patterns. For instance, a text source comprising of patterns or words occurring repeatedly. Besides, there are certain patterns where patterns are a rarity, instance being the *sgiomlaireached* word having less than a fraction probability of recurring in text sources. In such a situation, dictionary based approach, as observed earlier in the paper deems appropriation owing to its involvement in keeping references or list of repeated words (Sayood, 2012). Keeping the study aim in purview the Lempel-Ziv data compression algorithms has been discussed below.

LZ77 was proposed by Lempel and Ziv in 1977 as the first sequential based data compression algorithm which followed a dictionary-based approach. It is also known as LZ1 algorithm (Winters, Owsley, French, Bode, & Feeley, 1996), sometimes also referred to as "sliding window compression". In LZ77, the repeated occurrences are replaced with the references from the dictionary which were added while reading the original input. The encoder keeps the input in a sliding window while looking for matches. The sliding window comprises of two components: a search buffer which tracks the encoded data and a look-ahead buffer which contains the data about to be encoded (Ziv & Lempel, 1977). It has been quite popular for its use in Zip, PkZip, GNU gzip and LHarc which use LZ77 along with Huffman encoding for improving the compression ratio. Huffman encoding works on repetitive characters of a fixed length in the input data stream by re-assigning the smallest bit length with the highest frequently occurring character (Wolff & Papachristou, 2002). To overcome the weaknesses of LZ77, the above researchers, over the period of time, proposed an improved algorithm, LZ78 with a more flexible dictionary in 1978.

LZ78 was released by Lempel and Ziv in 1978 to fix the problems faced by developers while using LZ77. It is also known as LZ2 algorithm of data compression (Winters et al., 1996). LZ77 followed the concept of a text window while LZ78 followed a different approach for data compression. In LZ78 contains an unlimited list of previously encoded strings whereas the dictionary in LZ77 was composed of previously encoded text in a fixed-length window. LZ78 algorithm follows a simple mechanism by adding particular phrases to the dictionary and then, whenever those phrases repeat in the input stream again, they are replaced with the references from the dictionary, thus outputting a token made from the dictionary references instead of the phrases from the data stream reducing the size of the data. LZ78 also drops the need to supply the phrase length as a parameter to the decoder which used to be there in the case of LZ77 algorithm. LZ78 adds a new reference to the dictionary each time a token is output which was not there in the dictionary earlier, making it available to be referenced any time in the future. Thus, LZ78 works faster by creating dictionary on the fly, unlike LZ77 which used a dictionary of ready-made window full of text. LZ78 has also been quite popular for its use in Unix compact command and GIF image format for image compression (Ziv & Lempel, 1978).

Due to the success of LZ78 algorithm, it was adopted by Terry Welch to make an improved algorithm, LZW (Lempel–Ziv–Welch). With a simple and improved implementation, it was released in 1984 and became popular due to its high throughput in hardware implementations. After the LZ77, LZW also found its use in GIF image format for lossless image compression in 1987. It was also used in TIFF and PDF files at that time. LZW algorithm also started getting implemented in disk controllers with RAM required to support it. LZW also formed the core of Unix compression utility, compress (Wolff & Papachristou, 2002).

LZW algorithm for lossless data compression was created based on LZ78 algorithm. It uses a pre-initialized dictionary where all the possible characters are added with references at the start. Whenever a match is found, they are replaced with those references and if a match is not found in the input stream, then it is assumed to be the first character of an existing string in the dictionary, thus outputting only the last matching index. The algorithm was created mainly for hardware as the software hashing is marginally slower and less powerful than hardware hashing, and also the compression ratio depends on the hash calculation time carried out in the inner loop (Welch, 1984).

2.5 Design issues in compression

2.5.1 Issues while designing decaying dictionaries

Decaying dictionaries require an access or update each time a read or write operation is carried in Dynamic Frequent Value Cache technique, thus, making it an important part of the design. To make the dictionary design efficient for data cache compression, (Keramidas et al., 2006) proposes a dual port register file design for the DFVC technique. Another column of registers contains the decaying functionality and the local decay counter along with a decay status bit per entry signifying the current state. The functions performed by the counter and the status bit are recorded by decaying 4T memory cells. Another problem faced while designing the decaying dictionaries is that the dictionary requires adding a new frequent value to the first decayed entry, but searching the dictionary sequentially for the first decayed entry is not allowed as it makes the addition of a new value very slow and more expensive. This problem can be fixed by using a simple combinatorial circuit which can identify the initial decayed entry at the expense of a few gates.

2.5.2 Issues while designing Power-Aware DFVC (PA-DFVC)

In the model introduced by Yang and Gupta in their static dictionary implementation, the dynamic behavior of frequent values can be utilized in a power-aware compressed cache (Zhang et al., 2000). But Keramidas et al., (2006) and the fellow researchers overcome this issue with their implementation of dynamic dictionaries. In this technique, the frequent values are stored in Low Bit Array (LBA), whereas the non-frequent values are stored in both LBA and High Bit Array (HBA). The approach followed by them is that whenever a word is read from the cache, it is first read from the LBA which contain a flag bit used to direct to the next word. There is no requirement to read the HBA if the bit is set. Set bit means the bit was already stored there in the encoded form and then the technique proceeds to the process of decoding the value. But if that value is stored in un-encoded form, then only the HBA is read. When the read from LBA and HBA is arranged, it takes more time to attain a non-frequent value and it would be faster to read that particular from a conventional cache instead. Thus, the reduction of power consumption comes at the expense of an additional cycle spent on reading the non-frequent values.

2.5.3 Issues while designing High-Performance DFVC (HP-DFVC)

The static compression cache technique for L1 cache compression, proposed by (J. Yang et al., 2002), used a static dictionary in which either one compressed line or two compressed lines can be stored. But (Keramidas et al., 2006) improved the L1 cache compression with their dynamic cache compression technique by increasing its effective capacity. The cache and the dictionary contexts are kept constant and orderly in this technique. The phenomenon used assumes that either one compressed line or two compressed lines can be stored in each cache line of 2L. The line is kept in an uncompressed form if it is not possible to compress it to L words. Whereas, if the two compressed lines, after compressing both of them to L, refer to the same cache line, then the compressed lines can reside in that mapped cache line simultaneously. A flag bit is implemented to indicate whether a particular cache entry comprises of compressed lines. The cache entry also contains some useful information for the two compressed lines which is stored in tags (Tag1, Tag2), a valid bit and mask fields. The useful information about the two compressed lines is stored in the mask fields (Mask1, Mask2). If the tags are matched and the valid set bit is found, then the system determines a cache hit and moves to retrieving the word by inspecting the mask fields (Keramidas et al., 2006).

2.6 Review of dictionary-based techniques

There are few data compression techniques which work on a dictionary-based approach. LZ77 was the first Lempel-Ziv algorithm which brought a dictionary-based approach in light and it was proposed by Abraham Lempel and Jacob Ziv, in 1977 (Ziv & Lempel, 1977). Not so long after that, they improvised their LZ77 algorithm and released LZ78 algorithm in 1978 with a dictionary which could add new references on the fly, unlike the dictionary in LZ77 which was a ready-made window full of text (Ziv & Lempel, 1978). In 1982, James Storer and Thomas Szymanski came up with another lossless dictionary-based data compression algorithm which was based on LZ77. The major difference was that it would compare the size of reference with the original text to be replaced and replace it only if it meets the break-even point at least (Storer & Szymanski, 1982). Not so long after that, another algorithm based on LZ78 was introduced, Lempel-Ziv-Welch, by Terry Welch in 1984 (Welch, 1984). LZW was developed mainly for hardware implementations. It was even faster than LZ78 as it used a pre-initialized dictionary, unlike LZ78 which used to make dictionary instantaneously. Further, the research carried out by Ross Willams proposed LZRW in 1991 which although was based on LZ77 algorithm but used the literal/copy items mechanism of A1 algorithm by Fiala and Greene (Williams, 1991). In contemporary period, DEFLATE is a popular compression method which uses LZ77 algorithm along with Huffman coding to increase decompression speed and compression ratio (Mahmud, 2012).

2.7 Details of selected algorithm

As mentioned earlier, the LZRW1 was the first version of LZRW (Lempel-Ziv-Ross-Willams) series of algorithms proposed by Ross Williams in 1991 (Yang, Lekatsas, & Dick, 2006). As the name suggests, LZRW and its successor– LZRW1 falls in the class of LZ77 algorithms and inherits its advantages. It is built on the A1 algorithm which is also a member of LZ77 class of algorithms and was proposed by Fiala & Greene, in 1989. The LZRW1 algorithm follows the approach of single pass literal/copy from LZ77. The mechanism works by breaking the input stream into literal items and copy items which is done by using a single bit, known as control bit. The message is then represented in the form of literal and copy items. Copy items are used to represent the message as much as possible as they are the offsets and pointers to the strings already parsed in the history. If a match cannot be found in the dictionary, only then a literal item is used. Literal items represent the original uncompressed text (Reznik, 1998).

A literal item is composed of a single byte of data, while, a copy item comprises of two bytes of data which determine the length and offset of a string. The range for length and offsets are [3, 16] and [1, 4095] respectively. Control bits signify if a string is a literal item or a copy item. Control bits are congregated into groups of 16 to preserve byte alignment (Reznik, 1998).

The data structures of LZRW1 algorithm comprises of three parts– an input block, a hash table of 4096 pointers and some scalar variables. The hash table is the major component of the source model used in LZRW as it maps a three byte key to a single pointer which, in return, points to a matching key in the Lempel. An attempt to find a match for the initial three or more bytes in the Lempel of the Ziv at each step is made by the hash table. If a match is found in the Lempel, then a copy item is generated. The hash table is not required to be initialized in LZRW as the algorithm itself examines all the pointers that it gets from the hash table which, then, updates it after every item fetched rather than updating it after every byte. All of this makes the hash table update rate inversely proportional to the data compression. A copy item is made only if the pointer fetched points to the most recent 4095 bytes in the Lempel and also points to matching three bytes in the Ziv, otherwise a literal item is created. A copy item, thus created, represents the matched bytes, whereas, if a literal item is created, it has to represent only the first byte in the Ziv. The input block is present in the memory and it works as a read-only data structure in the LZRW1 technique (Reznik, 1998).

Using a simple hash table mechanism is what makes LZRW technique a fast text compression algorithm. LZRW1 text compression algorithm asks for around 13 machine instructions to carry the compression of each byte, whereas, 4 machine instructions to carry the decompression of each byte. LZRW1 compression runs about four times faster than the compress utility from Unix and is only 10% less efficient than that. The compress utility in Unix is based on LZC compression technique.

The researcher also tested it against the A1 algorithm and LZRW1 gave impressive results there. It turned out to be about ten times faster and only 4.3% less efficient than the A1 algorithm. These tests were done with a similar implementation on all compression algorithms and running them on a Pyramid 9820 computer (Lefurgy, Piccininni, & Mudge, 2000).

Another research carried out by Jakub Jaros, (2008) on "Word-based Dictionary Data Compression Methods" indulged in comparisons between various dictionary-based data compression algorithms and LZRW was one of them. Various dictionary-based algorithms were tested such as LZ77, LZSS, LZW, bzip, PPM and few others. The researcher carried out compression and decompression on all those algorithms by using the Calgary corpus, and Canterbury and Large Canterbury corpus data files. Same sets of files were used to compare LZRW with the Huffword2 compression technique. It turned out that the LZRW technique was faster and more efficient that Huffword2 algorithm by a very good margin. Huffword2 compression is a word-based model of Huffman coding.

LZRW1 Data Compression Algorithm
1. The first three bytes of Ziv are hashed.
2. Next step requires looking up the hash table yielding pointer p.
3. Then the table entry is replaced with pointer to Ziv
4. If the pointer p maps into Lempel, and the string corresponds to least first three bytes of Ziv, then the string is coded as a copy item, else, as a literal item.
5. At last, the Lempel/Ziv boundary is shifted.

Table 1: Process Structure of LZRW1 Algorithm (Williams, 1991)

Decompression, in the case of LZRW, is quite straightforward and swift. The decompressor handles a single item at a time and converts it into bytes. The translated bytes are attached to the end of the output. If the processed item is a literal item, then its single literal byte is attached, but if the processed item is a copy item, then the length and the offset fields of the copy item are used to locate it from the Lempel of the Ziv and copy a string which is already in the output block. During the whole process, control bits must be buffered (Williams, 1991).

2.8 Summary

Data compression is being used almost everywhere as the resources to store or transmit data are always limited and that is exactly where data compression comes into play. Finding the redundancy in data is the key to data compression whether it is lossy compression or lossless. Only the lossless compression is used when even a minimal loss of data can create problems for the organization. Although the series of Lempel-Ziv algorithms provide good lossless data compression, they have

their own weaknesses. Thus, they are usually used with other techniques such as Huffman coding or DEFLATE to improve compression or speed depending on the application. Data compression technique used in cache memory is also implemented at the cost of latency by increasing its apparent capacity. It can be implemented either with the use of a static dictionary or a dynamic dictionary but the designers can still face some issues while designing the memory. As mentioned above, data compression is all about redundancy. Replacing the repeated words with references to the index reduces the size of data. Thus, it can be said, more the redundancy in the data file, better compression ratio it can achieve.

Chapter 3: IMPLEMENTATION AND EVALUATION OF THE ALGORITHM

3.1 Introduction

So far, the basics of data compression and its techniques have been discussed along with issues faced while designing hardware for compression. A brief explanation has also been done for LZRW defining its whole process and components. LZRW, the algorithm used in this research, is a dictionary-based text compression algorithm and has been derived from LZ77 series of algorithms. It was developed by Ross Williams in 1991 and follows the lossless phenomenon of compression. It is a word-based text compression algorithm which divides the text into literal and copy items. Simply put, the literal items are those which appear in the input for the first time and the copy items are the repeated words (Williams, 1991). As the names suggest, higher the copy items, higher the compression ratio would be. Thus, the compression tries to use as much as much as copy items possible and use literal items only when there is no match found.

This chapter focuses on the implementation of LZRW1, the first version of LZRW series of algorithms. The algorithm has also been evaluated and compared with other algorithms such as LZW, LZO, PBPM, Bzip1, Gzip1 and many others. There are two important factors in compression: compression ratio and compression speed. All the tested algorithms have been evaluated on the basis of these two factors, wherever the data is available.

3.2 Implementation

The implementation code for the LZRW1 algorithm has been written in Java. A basic UI (User Interface) has also been made to take user's input with ease. The UI made uses Swing along with a slight usage of AWT. Only the basic elements have been added to the UI to keep it fast, light and clean. The screenshot below shows the UI.

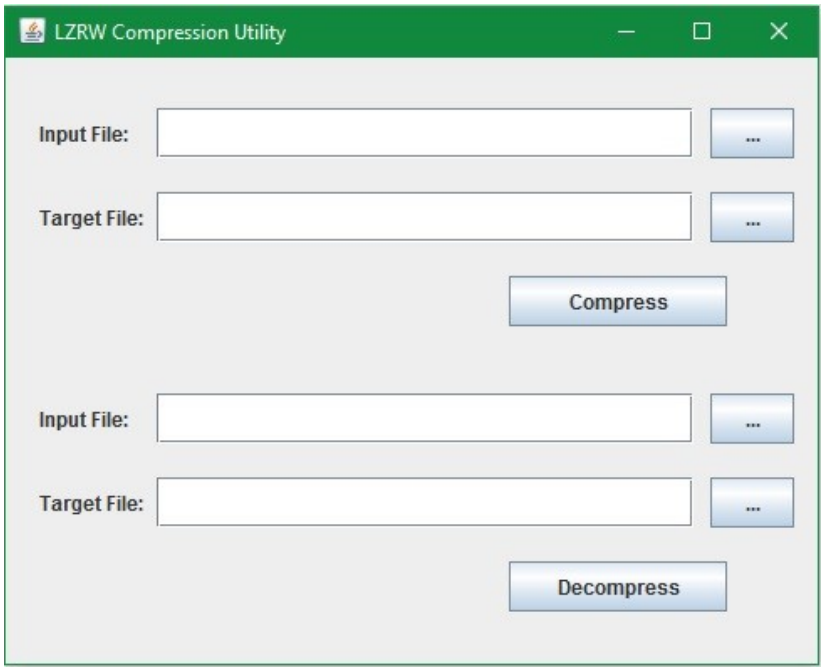


Figure 1: UI for developed algorithm

As it can be seen above, there is an option to take input file's address and output's file's address for compression and decompression, respectively. Both compression and decompression have been implemented in a single dialog box to keep the process fast and easy for the user. Clicking on the buttons next to the text areas open another dialog box where the input file or output address can be selected. The screenshot below (Figure 3) shows that dialog box in working.

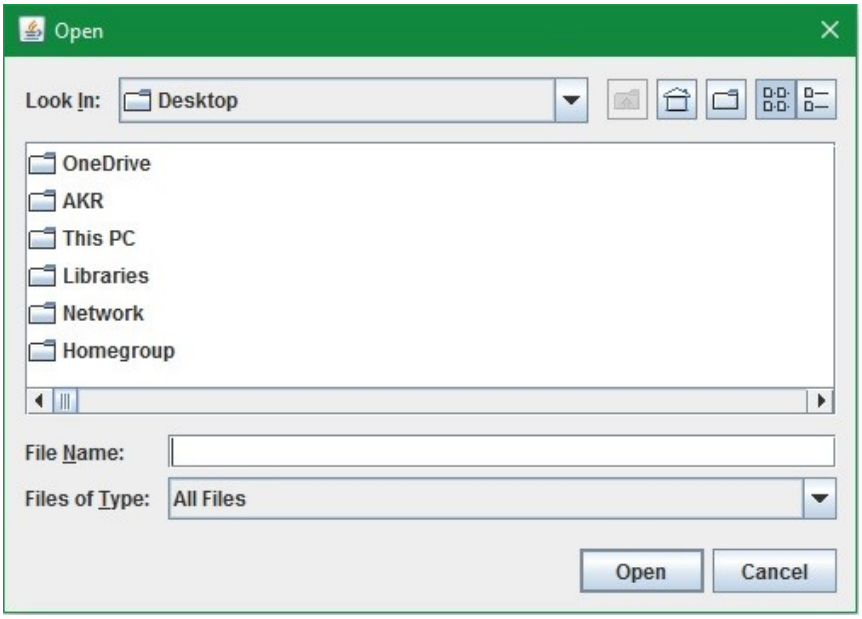


Figure 2: Dialogue box of compression and decompression

The input option for compression takes input as .txt files (text files) and gives output as .lzrw files. While the input option for decompression takes .lzrw files as input and gives output as .txt files. Once the input and output options have been selected, the user can click on the Compress/Decompress button to start the process. It takes less than a second for small files and output file is created in the target directory instantly. A confirm dialog box also appears asking "Are you sure you want to quit?", when you click on the close button.

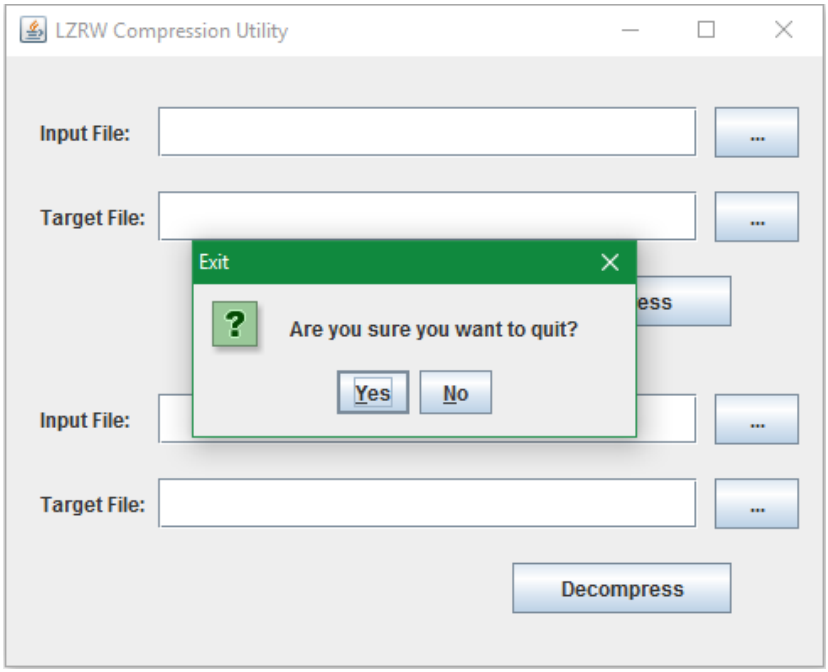
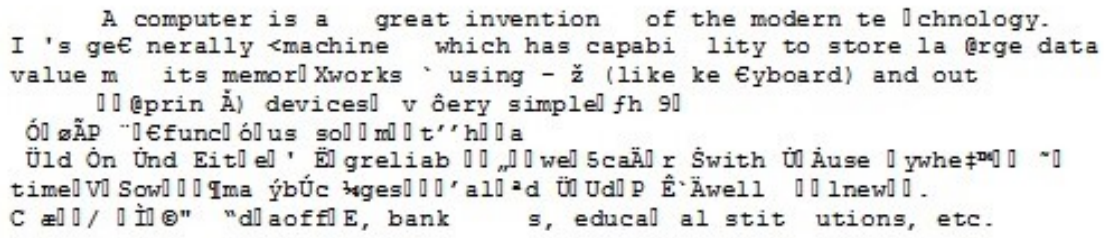


Figure 3: Confirmation window for closing output file

Sample strings with a variety of sizes were taken to test the algorithm and how good compression it is achieving. Screenshot of a sample string of 100 words, which was taken to test the compression ratio, is shown below.

```
A computer is a great invention of the modern
technology. It is generally a machine which has
capability to store large data value in its memory. It
works using input (like keyboard) and output (like
printer) devices. It is very simple to handle the
computer as its functioning is so common that a child
can handle it. It is a very reliable device which we
can carry with us and use anywhere and anytime. It
allows us to make changes in the already stored data
as well as store new data. Computer is a new
technology which is used in offices, banks,
educational institutions, etc.
```

The above text sample of about 100 words was used as input for the LZRW program, based on Java, which has been created during the research. The screenshot below shows the output in binary which came as a result of compression. Most of the part in the output was in binary, it was not encoded properly by the normal word processing softwares. Hence, the output screenshot may be missing few parts and should be considered only for preview purpose.



The input file, which contained the text sample of 100 words, had size of 580 bytes initially. Using the LZRW compression utility, created in this research, the input was compressed down to 480 bytes. It means that the compression achieved in this case was around 17.2%. After decompressing the compressed output, the utility successfully achieved the original file having the initial size of 580 bytes.

The Java program code running behind the process is not much complex and can be easily understood. It follows the algorithm published by Ross Williams in 1991 in the original LZRW research paper (Williams, 1991). The basic process of LZRW1 is as follows:

LZRW1 Data Compression Algorithm
1. The first three bytes of Ziv are hashed.
2. Next step requires looking up the hash table yielding pointer p.
3. Then the table entry is replaced with pointer to Ziv
4. If the pointer p maps into Lempel, and the string corresponds to least first three bytes of Ziv, then the string is coded as a copy item, else, as a literal item.
5. At last, the Lempel/Ziv boundary is shifted.

Table 2: Process for LZRW1 (Williams, 1991)

The “Lempel” referred to as in the algorithm is the history of items processed and “Ziv” is the next 16 bytes to be coded during the process. The process of selecting an item as a literal item or control item is done by a single bit here, known as control bit. Once the input item has been recognized as a literal item or a copy item, then the algorithm process follows.

The implementation of LZRW1 in Java also requires knowledge of bitwise operators which makes the implementation easier and keeps the code length small. As explained in the previous chapter, the LZRW1 compression consists of three major components: scalar variables, input block and a hash table of 4096 pointers. The hash table maps a three-byte key to a single pointer which, in return, points to a matching key in the Lempel. An attempt is made by the hash table to find a match for the initial three or more bytes in the Lempel of the Ziv at each step. If a match is found in the Lempel, then a copy item is generated.

The researcher, Ross Williams, published a C program code in the research paper showing how it can be implemented in C (Williams, 1991). Understanding that C program is necessary to implement the algorithm successfully as there is not much information given in the research paper about the decompression process. After understanding the original C program, it becomes easier to implement it in other languages. It has been implemented similarly in Java.

```
private void btComInputActionPerformed(java.awt.event.ActionEvent evt){ JFileChooser jfc =
new JFileChooser(FileSystemView.getFileSystemView().getHomeDirectory());
int returnValue = jfc.showOpenDialog(null);
if (returnValue == JFileChooser.APPROVE_OPTION) {
String str=  jfc.getSelectedFile().getAbsolutePath();
this.txtComInput.setText(str); }}
private void btComTargetActionPerformed(java.awt.event.ActionEvent evt) {
JFileChooser jfc = new
JFileChooser(FileSystemView.getFileSystemView().getHomeDirectory());
jfc.setAcceptAllFileFilterUsed(false);
FileNameExtensionFilter filter = new FileNameExtensionFilter("lzrw files", "lzr");
jfc.addChoosableFileFilter(filter);
int returnValue = jfc.showSaveDialog(null);
if (returnValue == JFileChooser.APPROVE_OPTION) {
String str=  jfc.getSelectedFile().getAbsolutePath();
if(!str.substring(str.length()-4, str.length()).equals(".lzr"))
str=str+".lzr";
this.txtComTarget.setText(str);
}}
```

Talking about the important parts of the Java program, the “btComInputActionPerformed” constructor in the java code, shown above, is responsible for taking the input for compression and

the “btComTargetActionPerformed” constructor works while selecting the output directory for the compression process.

```
private void btCompressActionPerformed(java.awt.event.ActionEvent evt) {
String infile=this.txtComInput.getText();
String outfile=this.txtComTarget.getText();
ByteArrayOutputStream data = new ByteArrayOutputStream();
if(infile.length()<=0)
{
int reply = JOptionPane.showConfirmDialog(LZRW.this,
"Input correctly.",
"Information",
JOptionPane.OK_OPTION,
JOptionPane.INFORMATION_MESSAGE);}
FileInputStream in;
try {
in = new FileInputStream(infile);
int b = in.read();
while( b >=0 ){
data.write((byte)b);
b = in.read();
}
in.close();
byte[] tmp=data.toByteArray();
int[] inBytes = new int[tmp.length+1];
for(int i=0;i<tmp.length;i++)
{inBytes[i]=tmp[i];}
int[] outBytes = new int[inBytes.length*2];
int compressedLen = compress(inBytes,outBytes);
int[] temp = new int[compressedLen];
System.arraycopy(outBytes,0,temp,0,compressedLen);
outBytes = temp;
tmp=new byte[compressedLen];
for(int i=0;i<tmp.length;i++)
{tmp[i]=(byte)temp[i]; }
FileOutputStream out = new FileOutputStream(outfile);
```

```
out.write(tmp);
out.close();
} catch (IOException err) {}
}
```

The “btCompressActionPerformed” constructor above shows the handling of input file done by the algorithm. It gets text from the input text file, processes it using the “compress” method and sends the output to the output .lzrw file at the target directory selected by the user. Below is the “compress” method for the process of LZRW1 compression based on the program made by the researcher. In the following code, the code selects if an item is a literal item or a copy item which is done using a control bit, as explained in the algorithm earlier.

```
public int compress(int[] src, int[] dst ){
int isrc = 0, idst = 0;
int isrc_max1=src.length-ITEMMAX, isrc_max16=src.length-16*ITEMMAX;
int[] hash = new int[4096];
int icontrol;
int control=0,control_bits=0;
dst[idst]=FLAG_COMPRESS;
idst+=FLAG_BYTES; icontrol=idst; idst+=2;
boolean overrun = false;
while (true){
int p=0,s=0;
int unroll=16,len,index;
int offset=0;
if ( idst>dst.length) {
overrun = true;
break;
}
boolean literal = false;
if (isrc>isrc_max16){
unroll=1;
if (isrc>isrc_max1){
if (isrc==src.length) break;
literal = true;
}}
do{
```

```
if(!literal ){
index=((40543*((((src[isrc]<<4)^src[isrc+1])<<4)^src[isrc+2]))>>4) & 0xFFF;
p=hash[index]; hash[index]=s=isrc; offset=s-p;
}
if( literal || offset>4095 || p<0 || offset==0 || src[p++] != src[s++]
|| src[p++] != src[s++] || src[p++] != src[s++])
{
literal = false;
dst[idst++]=src[isrc++];
control>>=1;
control_bits++;
}
else
{
boolean foo = src[p++] != src[s++] || src[p++] != src[s++] || src[p++] != src[s++] || src[p++] !=
src[s++]
|| src[p++] != src[s++] || src[p++] != src[s++] || src[p++] != src[s++] || src[p++] != src[s++]
|| src[p++] != src[s++] || src[p++] != src[s++] || src[p++] != src[s++] || src[p++] != src[s++]
|| src[p++] != src[s++] || (s++ != 0);
len=s-isrc-1;
dst[idst++]=(((offset&0xF00)>>4)+(len-1));
dst[idst++]=(offset&0xFF);
isrc+=len; control=(control>>1)|0x8000; control_bits++;
}
}while(--unroll != 0);
if (control_bits==16)
{
dst[icontrol]=(control&0xFF);
dst[icontrol+1]=(control>>8);
icontrol=idst; idst+=2; control=control_bits=0;
}}
if( overrun ){
System.arraycopy(src,0,dst,FLAG_BYTES,src.length);
dst[0]=FLAG_COPY;
return src.length+FLAG_BYTES;
}
```

```
control>>=16-control_bits;
dst[icontrol++]=(byte)(control&0xFF);
dst[icontrol++]=(byte)(control>>8);
if (icontrol==idst)
idst-=2;
return idst;
}
```

The code above shows the method responsible for the compression process where the input block has been specified using p_src_first and src_len. The p_dst_first is pointed to the output zone and p_dst_len is pointed to an unsigned long of 4 bytes to receive output length. After that, the length of the output block is written to *p_dst_len.

The “compress” method keeps the hash table up to date by replacing the already fetched entry with a new entry from the Ziv, in case of a literal item. But it is a copy item, if it maps a three-byte key to point to that word’s recent occurrence. After compression, following things are stored in the output file, literal or copy items along with control bit, offset and length of copy items. They are later used during the decompression process.

```
public int decompress(int[] src, int[] dst){
int controlbits=0, control=0;
int isrc=FLAG_BYTES, idst=0;
if (src[0]==FLAG_COPY){
System.arraycopy(src,FLAG_BYTES,dst,0,Math.min(src.length-FLAG_BYTES,dst.length));
return src.length-FLAG_BYTES;
}
while (isrc != src.length){
if (controlbits==0){
control=src[isrc++];
control|=(src[isrc++])<<8;
controlbits=16;
}
if ((control & 1) > 0){
int offset,len; int ip;
offset=(src[isrc]&0xF0)<<4;
len=1+(src[isrc++]&0xF);
offset+=src[isrc++]&0xFF;
```

```
ip=idst-offset;
while (len-- != 0){
dst[idst++]=dst[ip++];
}}
else{
dst[idst++]=src[isrc++];
}
control>>=1;
controlbits--;
}
return idst;
}
```

The “decompress” method (shown above) contains the code for the decompression process. It starts with the control bit which was stored during the compression process. During the decompression phase, it first checks whether the control bit is 0 or 1 on the basic of which it decides whether the item is literal or copy. If it literal item, then it appends the item from source into the output as it is. But if it is the literal item, then it reads the offset and length which were also stored during the compression. Using the offset and length value, it goes back and reads the compressed file and appends it to the destination file.

Data set:

For further test of the LZRW1 we applied to four groups of files classified according to the types of the files in the group (the Canterbury corpus, the artificial corpus, the large corpus, and miscellaneous corpus) (http://corpus.canterbury.ac.nz/descriptions/) the result of the test shown bellow:

Test one: the Canterbury corpus contain four files with different sizes as shown on table 3 and table 4 we use compare between LZRW1 code and WINZIP tool.

File Name	File size in bytes	Description	File source	File size after compression in bytes	Compression ratio	saving space
asyoulik.txt	125179	Shakespeare	Canterbury Corpus	46766	2.99	62.64%
alice29.txt	152089	English text	Canterbury Corpus	51858	2.73	65.9%
lcet10.txt	426754	Technical writing	Canterbury Corpus	139142	2.61	67.4%
plrabn12.txt	481861	Poetry	Canterbury Corpus	186098	3.09	61.38%

Table 3. Results for different English files using WinZip

File Name	File size in bytes	Description	File source	File size after compression in bytes	Compression ratio	saving space
asyoulik.txt	125179	Shakespeare	Canterbury Corpus	80466	5.14	35.72%
alice29.txt	152089	English text	Canterbury Corpus	93371	4.91	38.6%
lcet10.txt	426754	Technical writing	Canterbury Corpus	253980	4.76	40.48%
plrabn12.txt	481861	Poetry	Canterbury Corpus	326269	5.42	32.29%

Table 4. Results for different English files using LZRW1 code

The following graph shows that the efficiency of the WINZIP tool better than the LZRW1

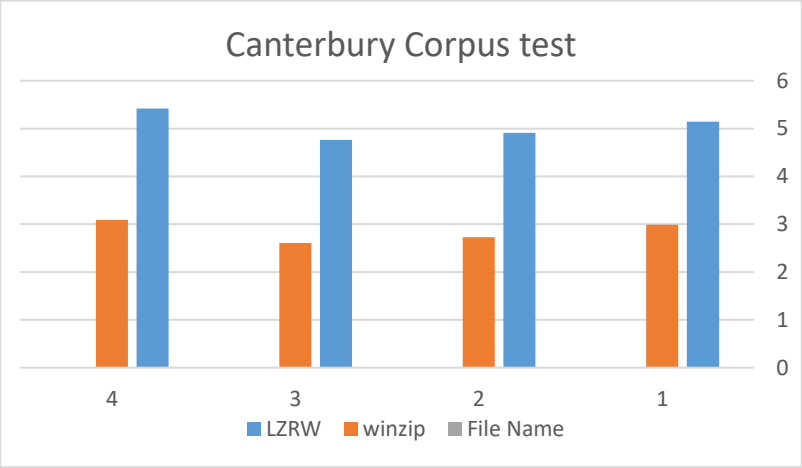


Figure 4: Compression ratio for WINZIP and LZRW1 Code

Test two: the artificial corpus contains two files with same sizes as shown on table 5 and table 6 we use compare between LZRW1 code and WINZIP tool.

File Name	File size in bytes	Description	File source	File size after compression in bytes	Compression ratio	saving space
aaa.txt	100,000	The letter 'a', repeated 100,000 times.	Canterbury Corpus	275	0.02	99.73%
alphabet.txt	100,000	Enough repetitions of the alphabet to fill 100,000 characters	Canterbury Corpus	454	0.04	99.55%

Table 5. Results for different English files using WinZip

File Name	File size in bytes	Description	File source	File size after compression in bytes	Compression ratio	saving space
aaa.txt	100,000	The letter 'a', repeated 100,000 times.	Canterbury Corpus	13288	1.06	86.71%
alphabet.txt	100,000	Enough repetitions of the alphabet to fill 100,000 characters	Canterbury Corpus	13319	1.07	86.68%

Table 6. Results for different English files using LZRW1 code

The following graph shows that the efficiency of the WINZIP tool better than the LZRW1.

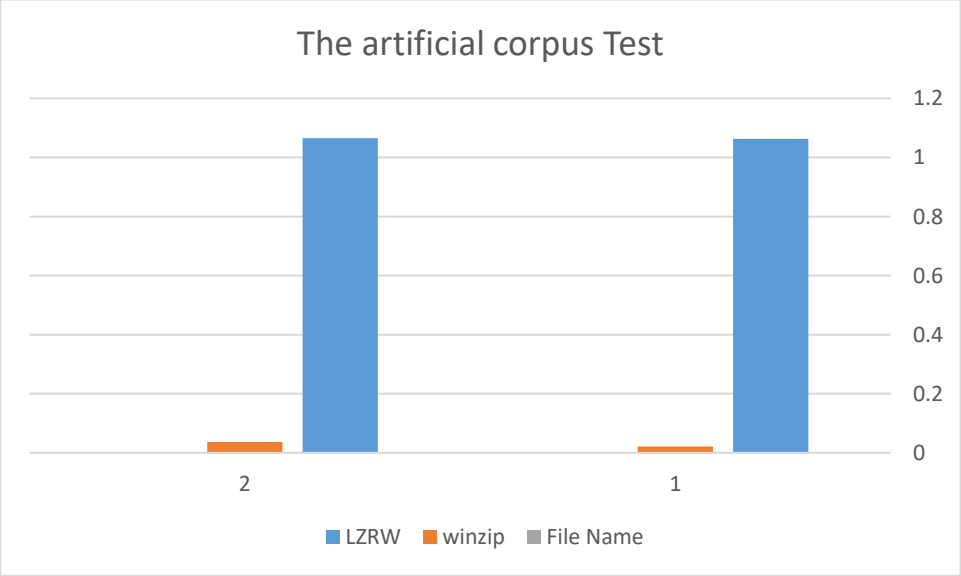


Figure 5: Compression ratio for WINZIP and LZRW1 Code

Test three: the large corpus contains two files with different sizes as shown on table 7 and table 8 we use compare between LZRW1 code and WINZIP tool.

File Name	File size in bytes	Description	File source	File size after compression in bytes	Compression ratio	saving space
bible.txt	4047392	The King James version of the bible	Canterbury Corpus	1121158	2.22	72.30%
world192.txt	2473400	The CIA world fact book	Canterbury Corpus	709790	2.30	72.30%

Table 7. Results for different English files using WinZip

File Name	File size in bytes	Description	File source	File size after compression in bytes	Compression ratio	saving space
bible.txt	4047392	The King James version of the bible	Canterbury Corpus	1121158	4.33	45.82%
world192.txt	2473400	The CIA world fact book	Canterbury Corpus	709790	4.99	37.51%

Table 8. Results for different English files using LZRW1

The following graph shows that the efficiency of the WINZIP tool better than the LZRW1

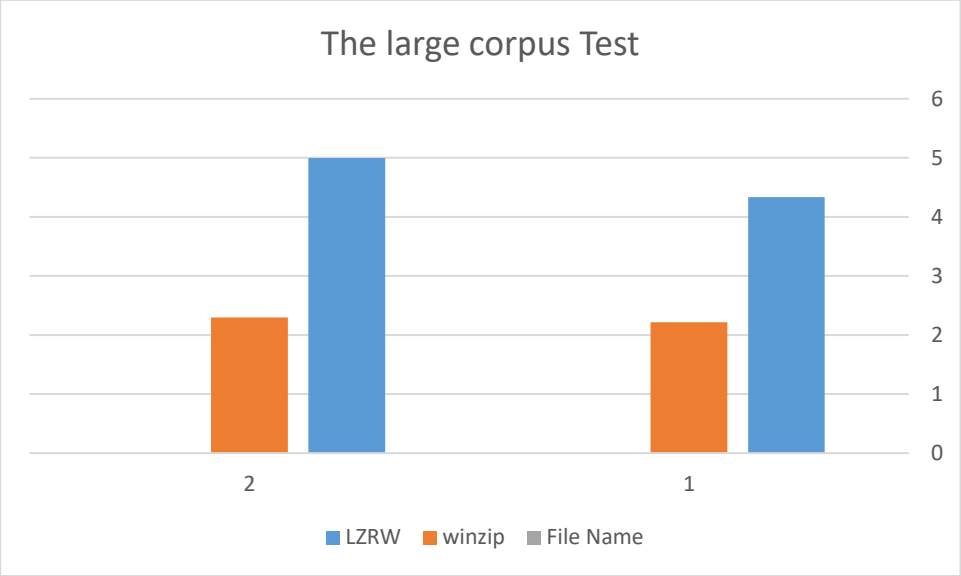


Figure 6: Compression ratio for WINZIP and LZRW1 Code

Test four: The Miscellaneous Corpus contains one file as shown on table 9 and table 10 we use compare between LZRW1 code and WINZIP tool.

File Name	File size in bytes	Description	File source	File size after compression in bytes	Compression ratio	saving space
pi.txt	4047392	The first million digits of pi	Canterbury Corpus	1121158	3.66	54.24%

Table 9. Results for different English files using WinZip

File Name	File size in bytes	Description	File source	File size after compression in bytes	Compression ratio	saving space
pi.txt	4047392	The first million digits of pi	Canterbury Corpus	1121158	5.78	27.71%

Table 10. Results for different English files using LZRW1

The following graph shows that the efficiency of the WINZIP tool better than the LZRW1

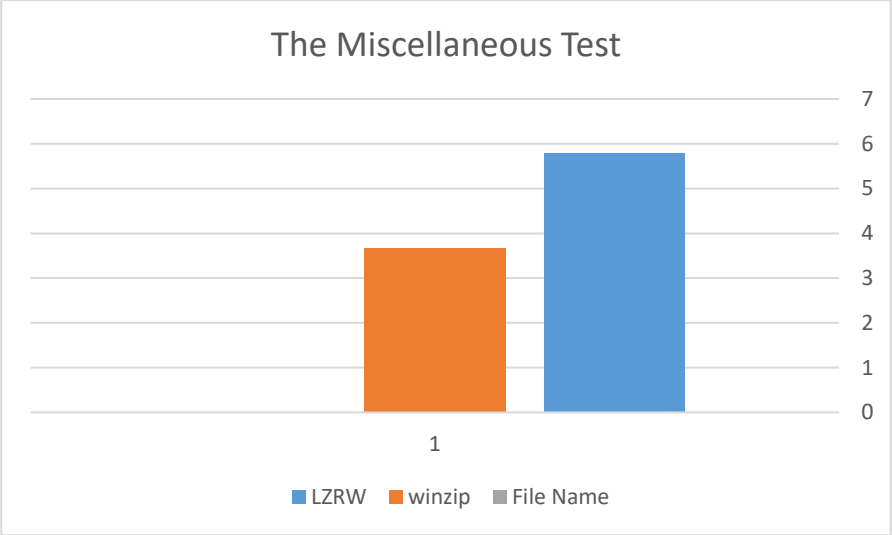


Figure 7: Compression ratio for WINZIP and LZRW1 Code

3.3 Experimental evaluation

As mentioned earlier in this research, LZRW appeared in 1991 with an aim to provide best text compression. It was indeed faster than other text compression dictionary based algorithms available at that time. Since then, it has been evaluated and compared with other compression algorithms numerous times by various researchers all over the globe. Few of their studies will be taken here to compare LZRW with algorithms released before it and some of the modern compression algorithms.

Various evaluation tests were performed by the researcher, Ross Williams, in his research paper "An Extremely Fast ZIV-Lempel Data Compression Algorithm" before releasing LZRW to the

general public (Williams, 1991). During the tests, it was implemented in both high level and low-level languages to test the performance, compression ratio and running times. The algorithms, which LZRW1 was compared with, were LZC algorithm and A1 algorithm. First, it was tested on a Pyramid 9820 computer system running on Unix operating system with similar implementations of all the algorithms in C. A standard corpus of test case files were used to evaluate the algorithms. As a result, LZRW1 turned out to be 10% than LZC algorithm in terms of compression ratio, but 4 times faster in terms of speed. When it was compared to A1 algorithm on the system, LZRW1 resulted in 4.3% worse compression ratio than A1 algorithm, but ran 10 times faster. It concluded that the exhaustive search of A1 algorithm for processed items and its two-byte usage of minimum copy length didn't improve its compression to a great extent.

Algorithms	**Compression (%Rem)**	**Compression Speed**	**Decompression Speed**
LZRW1	55.5	224	394
LZC	45.5	58	94
A1	51.2	22	429

Table 11. Average Results (Williams, 1991)

The above table shows average results of the tests performed by the researcher. The Compression (%Rem) denotes the compression as percentage remaining, lower is better. The compression and decompression speeds are kilobytes/sec. It was also found out that LZRW gave relatively poor compression performance in case of English text files when compared to non-English text files.

In a 1996 research paper "LZP: A New Data Compression Algorithm" by Charles Bloom, the researcher tested the LZRW algorithm with several variants of LZP compression algorithm (Bloom, 1996). The evaluation tests were run on an Amiga 3000 computer system which was using a Motorola 68030 processor clocked at 25 MHz. Test case files of Calgary Corpus were used for the tests. Speeds were reported in bytes were second and the compression ratio was recorded in bits per byte. Disk access times were excluded from the recorded compression and decompression times. LZRW gave more than 7% better compression than the various implementations of LZP at an average, but was beaten by LZS by around 3% under the same terms. LZRW was about 27% faster than LZP1 implementation and also consumed less space in the memory. While the second

implementation of LZP was a lot faster than LZRW1, but it again got beaten by LZRW1 in terms of compression ratio.

In another research paper "CRAMES: Compressed RAM for Embedded Systems" by Lei Yang and other fellow researchers in 2005, LZRW1 was evaluated with bzip2, zlib (default, level 1 and level 9), RLE (Run Length Encoding) and LZO for memory compression in embedded systems (Yang, Dick, Lekatsas, & Chakradhar, 2005). The research was done to find a memory compression algorithm with good performance and energy efficiency, low compression and low memory overhead. The tests were performed on embedded systems which require both on-line data memory compression and in-RAM filesystem compression. The resulted memory overheads are as follows:

Algorithms	**Compression**	**Decompression**
bzip2	7600 kB	3700 kB
Zlib	256 KB	44 KB
LZO	64 KB	0
LZRW1-A	16 KB	16 KB
RLE	0	0

Table 12 Memory overheads (CRAMES: Compressed RAM for Embedded Systems)

The memory overheads were very high in case of bzip2 and zlib, whereas the LZRW gave acceptable memory overheads. RLE turned out to be the best of all. It had high compression ratio and low compression/decompression times. LZRW stood next to RLE in the test results. LZRW had an average compression ratio of about 0.30. It easily beat bzip2, zlib (default, level 1 and level 9) and LZO with ease in terms of compression/decompression times. LZO was selected for its optimal performance in block compression in low-power embedded systems due to its efficiency. LZO has low memory requirements for compression and requires no memory in case of decompression which suited well with the embedded systems.

Few more tests were conducted by Jakub Jaros in 2008 when various dictionary-based data compression algorithms LZ77, LZSS, LZW, bzip, PPMd, LZRW and many others were compared with each other (Jaros, 2008). All the algorithms taken for tests were word-based, hence, the research was named as "Word-based Dictionary Data Compression Methods". In those tests, LZRW1 algorithm was compared with Huffword2 which is a word-based model of Huffman coding data compression algorithm. Testing system was running on a AMD Athlon 3200 processor clocked at 2.2 GHz. The evaluation tests were carried out with Calgary corpus, and Canterbury and Large Canterbury corpus data files and all the algorithms were coded in C++ programming language. When compared with adaptive word-based models of LZW, gzip-9 and ppmD7, LZRW1 resulted in providing the worst compression ratio. Similar results were obtained when LZRW was compared with Huffword2 using the Calgary corpus, and Canterbury and Large Canterbury corpus text files. Huffword2 gave better compression results, while LZRW1 was the faster one out of the two.

In a 2010 research regarding "Fast Text Compression Using Multiple Static Dictionaries" done by A. Carus and A. Mesut, LZRW was again tested against various dictionary-based compression algorithms such as LZW, PPMd, Bzip-1, Gzip-1, WRT, STECA and LZP1 (Carus & Mesut, 2010). A computer system with an Intel Core 2 Duo 1.83 GHz processor, 2GB of RAM, running with Microsoft XP operating system, was used to run the evaluation tests. The implementations of algorithms were done in C and were run on Microsoft Visual Studio. Text files in two languages: English and Turkish were used in the compression tests with a total size of 16,453,299 bytes and 15,828,594 bytes respectively. The compression ratios were recorded in bits per character and compression/decompression speeds were recorded in Mbps. To record the compression and decompression speeds, the text files were compressed and decompressed 6 times and their average was taken.

Algorithms	Compression Ratio (bpc)	Compression Speed (Mbps)	Decompression Speed (Mbps)
PPMd	1.89	86	73
Bzip-1	2.43	33	92
Gzip-1	3.43	132	369
WRT-0	3.65	153	148
STECA-DT	3.69	273	405
STECA-T	4.03	314	418
STECA-D	4.21	433	448
LZW	4.33	141	273
LZOP-1	4.44	299	546
LZP-1	4.86	209	177
LZRW1	4.96	209	216

Table 13 Results for English Test Files (Fast Text Compression Using Multiple Static Dictionaries 2010)

As it can be seen in the table above that PPMd compression algorithm topped the charts in terms of compression ratio but it lagged behind in terms of speed. Bzip and Gzip took the second and third position in compression ratio, while, the LZRW1 algorithm turned to be the worst. The maximum speed achieved during compression was 433 Mbps and during decompression, it surpassed to 546 Mbps. Looking at the compression and decompression results, it can be observed that STECA-D

turned out to be the fastest in compression, whereas, LZOP-1 topped the decompression results followed by STECA-D and STECA-T at second and third place respectively. PPMd, which topped the charts in terms of compression ratio, was at the bottom in decompression speed results. During these tests, LZRW1 algorithm gave worst compression results, but its compression/decompression speeds were acceptable. It is quite common to see the algorithm with best compression to give worst speed results which can be seen in these test results also.

Research, Year	**Algorithm**	**Hardware Used**	**Compression Ratio**	**Compression Speed/Time**	**Decompression Speed/Time**
Fast Text Compression Using Multiple Static Dictionaries, 2010	LZRW1	Intel Core 2 Duo 1.83 GHz	4.96 bpc	209 Mbps	216 Mbps
Fast Text Compression Using Multiple Static Dictionaries, 2010	LZW	Intel Core 2 Duo 1.83 GHz	4.33 bpc	141 Mbps	273 Mbps
Fast Text Compression Using Multiple Static Dictionaries, 2010	LZP	Intel Core 2 Duo 1.83 GHz	4.86 bpc	209 Mbps	177 Mbps
Fast Text Compression Using Multiple Static Dictionaries, 2010	STECA-D	Intel Core 2 Duo 1.83 GHz	4.21 bpc	433 Mbps	448 Mbps

Fast Text Compression Using Multiple Static Dictionaries, 2010	Gzip	Intel Core 2 Duo 1.83 GHz	3.43 bpc	132 Mbps	369 Mbps
Fast Text Compression Using Multiple Static Dictionaries, 2010	Bzip	Intel Core 2 Duo 1.83 GHz	2.43 bpc	33 Mbps	92 Mbps
Fast Text Compression Using Multiple Static Dictionaries, 2010	PPMd	Intel Core 2 Duo 1.83 GHz	1.89 bpc	86 Mbps	73 Mbps
Word-based Dictionary Data Compression Methods, 2008	huffword2	AMD Athlon 3200 2.2 GHz	4.37 bpc	3.47 seconds	0.84 seconds
CRAMES: Compressed RAM for Embedded Systems, 2005	RLE	HP iPAQ hx2755	2.96 bpc	0.0001 seconds	0.0001 seconds
CRAMES: Compressed RAM for Embedded Systems, 2005	LZO	HP iPAQ hx2755	1.84 bpc	0.0002 seconds	0.0001 seconds

Table 14 Results for English Test Files (Fast Text Compression Using Multiple Static Dictionaries 2010)

3.4 Summary

While the LZRW1 implementations used in evaluation tests were in C and C++, the code implementation done in this research is in Java programming language. The code developed in Java

follows the footsteps of the original C program of LZRW1 developed by Ross Williams. The Java implemented program comes with a basic UI which is easy to use and doesn't waste unnecessary resources. All it asks for is just the Java Runtime Environment installed on any computer system.

Since, the launch of LZRW algorithm, numerous of dictionary-based data compression algorithms have been released. Many of them provide better compression ratios than LZRW1, but then they lag behind LZRW1 in terms of compression and decompression speeds. In this chapter, LZRW1 (the first version of LZRW) was compared with the following algorithms: LZC, A1, LZP, bzip, gzip, zlib, LZO, RLE, PPMd, LZ77, LZSS, Huffman coding and many others with large text files running on variety of computer or embedded systems. All evaluation tests gave similar results. After 26 years, it can still beat several modern text compression algorithms in terms of speed while giving optimal compression ratios in some cases. It all depends on the usage requirement after all.

Chapter 4: CONCLUSION AND RECOMMENDATIONS

This research was based on dictionary-based data compression algorithms out of which LZRW was selected. In the starting chapters, various basic topics have been covered regarding dictionary-based data compression. Then, there are the design issues, which are usually faced by the hardware designers while implementing data compression in cache memory, very well explained followed by the solutions to those design issues. LZRW, the main focus of this research, has been discussed with a brief explanation of the algorithmic process and the data structure components used in the LZRW algorithm such as scalar variables, input block and hash table. A total of 7 versions of LZRW were released by Ross Williams in 1991. The variant of LZRW used in this research is LZRW1. It was the first version released with a basic implementation. Versions released after LZRW1 came with slight changes making it more efficient and fast.

In the previous chapter, a Java program in action running on LZRW compression technique, made for this research only, was presented. The implementation written along with the Java program explains the code very well and shows how the code handles the input and gives an output, how various constructors provide the UI using swing and AWT and how the methods work during the compression and decompression. The evaluation tests taken from several researches show the performance and efficiency of LZRW1 compression algorithm, although most of the researches have used C implementations of LZRW algorithm and the program in this research has been developed using Java. The evaluation tests clearly provide the comparison of LZRW with other data compression algorithms, including some modern ones too such as bzip, gzip, zlib, Huffman coding, RLE and many others.

This chapter focuses on talking about various things which has been observed during the research, the things concluded after the thorough study of LZRW and its evaluation results. The recommendations will be provided on the basis of the study that has been performed so far during the research. A talk about the future scope of LZRW and this research will also be carried out after recommendations and summary.

4.1 Conclusion

The scope of this study was dictionary-based data compression techniques. There can be static dictionaries used or dynamic dictionaries. This research focused on static dictionary-based approaches while there are dynamic dictionary-based approaches also. Dynamic dictionary-based approaches were out of the scope of this research, but there has been a slight introduction given about them after briefly describing what exactly the data compression is, its various types, what are the various techniques used in data compression and what is dictionary-based compression. There are many design issues faced by hardware designers who implement data compression in cache memory using static dictionary based approaches and that slight introduction, regarding dynamic dictionary based approaches, provides solutions to those issues (Keramidas, Aisopos, & Kaxiras, 2006). Those issues occur while designing the decaying dictionaries, Power-Aware DFVC (PA-DFVC) and High-Performance DFVC (HP-DFVC). The main advantage of dynamic dictionaries is that they update by themselves on-the-fly. Various available dictionary-based data compression algorithms in this research, such as LZ77, LZ78, LZW and other Lempel-Ziv series of data compression algorithms depicted how do dictionary-based algorithms work and how do they vary from each other. Studying other dictionary-based algorithms helped understand various the benefits LZRW has over the other algorithms. The use of hash table and a control bit to decide whether an item is literal or copy is what makes the LZRW compression algorithm faster than the other dictionary-based compression algorithms.

The main objective of this research was to study the LZRW in depth and there has been a thorough research done here in that regard. Everything about LZRW has been very well explained in previous chapters. Its basics, the mechanism included, its algorithmic procedure, the data structure components used in the compression and decompression processes, everything has been covered in this research along with some evaluation tests from various researches. The algorithmic process of LZRW is a five-step procedure with a little tricky implementation, but the C program from the original research paper does help a lot (Hankerson, Johnson, & Harris, 1998) . The compression process has been clearly described in the research paper which makes it easier for a developer to understand it and implement it. But, there is nothing mentioned about decompression process in the research paper, so one has to be able understand the C program given to understand the decompression process clearly and to know how it can be implemented. Once the developer has understood the C program implementing it or porting the program to any other programming language is relatively simpler. After writing the code for compression, the decompression process takes just a small piece of code which can be seen in previous chapter under the implementation topic.

The implementation of LZRW compression was done using Java programming language in this research because of its easy portability and fast implementation. Also, java is easier to understand and code which makes it painless for other developers to fix or extend the code in future (IBM, 2014). Only thing that programs written in Java require is JRE (Java Runtime Environment), regardless of the configuration of the host computer system is. The program was developed with a clean user-friendly interface making it easier for the end-user to interact with the software comfortably. The end-user would need to possess little technical knowledge to use the LZRW compression program. Everything about the implementation process has already been explained in the previous chapter which includes all the information about the major constructors and methods handling the UI, compression and decompression processes. The evaluation tests of LZRW with other dictionary-based data compression algorithms gave an idea about where does it stand with modern algorithms. The algorithms with which LZRW was compared also included algorithms introduced in 1980-1990s such as LZ77, LZW, LZS and many other dictionary-based text compression algorithms. The evaluation tests were performed using implementation of algorithms done in C language. The evaluation tests were done on normal computer systems as well as some embedded systems and large text files in multiple languages were used in some tests. The results suggested that the compression in LZRW was quite faster in most of the test cases. It even beat some modern text compression algorithms in terms of compression speed, but the compression ratio was not up to the mark to compete with those modern algorithms such as Huffman coding, RLE, zlib and others. It was able to beat the older algorithms in compression ratio and compression speeds.

4.2 Recommendations

The evaluation tests and its comparisons with other data compression algorithms have clearly shown that the LZRW text compression is not much effective in this age. It was released 26 years ago and a whole lot has changed in those years. There are several modern lossless data compression algorithms which can do better compression and decompression with faster speeds such as Huffman coding, RLE and few others which are quite popular. Lots of variations of LZ77 algorithms have been released since then which are better and faster. Nowadays, multiple algorithms are used together to get best possible compression ratios and speeds and DEFLATE is a good example of that. DEFLATE is based on LZ77 and Huffman coding algorithms used together and the algorithm is quite popular for its use in Zip compression, PNG files and PDF files. Thus, the modern algorithms such as DEFLATE, bzip2, PPMd, bzip2 and many other modern data compression algorithms can be used instead of LZRW to achieve better compression. More can be recommended

depending on the usage requirements of the user. Each algorithm comes with its own benefits and disadvantages. Hence, depending on the user's requirements, the most suitable algorithm can be chosen whether it is about achieving faster compression, less consumption of system resources or it is about attaining best compression ratios.

4.3 Summary and Future Scope

This research regarding dictionary-based algorithms and LZRW algorithm provides a brief covering all their aspects. The implementation of LZRW written in Java programming language is very easy to understand. The evaluation tests of LZRW with other old and new algorithms show more than enough about where does LZRW stand in this modern age of data compression algorithms. The algorithm has become outdated and there are lots of algorithms which have been released since LZRW' release. The user should rather opt for those algorithms if achieving best compression ratio is the major target of the user. LZRW gives low memory overhead and has still got an acceptable compression speed and even faster in some cases. So, in the end, it all depends on the requirements of the user which algorithm he should opt for. The java program can also be modified according to user's needs and opinions in future to make the compression process more effective and faster.

REFERENCES

Alameldeen, A. R., & Wood, D. A. (2004). Adaptive cache compression for high-performance processors. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.* (pp. 212–223). IEEE. https://doi.org/10.1109/ISCA.2004.1310776

Blelloch, G. E. (1998). Introduction to Data Compression. In G. E. Blelloch (Ed.), *Algorithms in the Real World* (pp. 1–29). School of Computer Science, Carnegie Mellon University.

Bloom, Charles. "LZP: A New Data Compression Algorithm." Data Compression Conference. 1996.

Carus, A., and A. Mesut. "Fast text compression using multiple static dictionaries." Information Technology Journal 9.5 (2010): 1013-1021.

Fiala, E. R., & Greene, D. H. (1989). Data compression with finite windows. *Communications of the ACM*, *32*(4), 490–505. https://doi.org/10.1145/63334.63341

Jaros, J. (2008). Word-based Dictionary Data Compression Methods, (May).

Keramidas, G., Aisopos, K., & Kaxiras, S. (2006). Dynamic Dictionary-Based Data Compression for Level-1 Caches.

Lefurgy, C., Piccininni, E., & Mudge, T. (2000). Reducing code size with run-time decompression. In *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)* (pp. 218–228). IEEE Comput. Soc. https://doi.org/10.1109/HPCA.2000.824352

Lefurgy, Charles, Eva Piccininni, and Trevor Mudge. "Reducing code size with run-time decompression." *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on*. IEEE, 2000.

Lekatsas, Haris, et al. "CRAMES: compressed RAM for embedded systems." Hardware/Software Codesign and System Synthesis, 2005. CODES+ ISSS'05. Third IEEE/ACM/IFIP International Conference on. IEEE, 2005.

Lelewer, D. A., & Hirschberg, D. S. (1987). Data Compression. *ACM Comput. Surv.*, *19*(3), 261.

Mahmud, S. (2012). An Improved Data Compression Method for General Data. *International Journal of Scientific & Engineering Research*, *3*(3). Retrieved from

http://www.ijser.org/paper/An-Improved-Data-Compression-Method-for-General-Data.html

Nelson, M., & Gailly, J. (1995). *The Data Compression Book* (2nd ed.). Wiley.

Reznik, Y. A. (1998). LZRW1 without hashing. In *Proceedings DCC '98 Data Compression Conference (Cat. No.98TB100225)* (p. 569). IEEE Comput. Soc. https://doi.org/10.1109/DCC.1998.672311

Salomon, D., & Motta, G. (2010). *Handbook of data compression* (5th ed.). Springer. Retrieved from https://books.google.co.in/books?id=LHCY4VbiFqAC&dq=data+compression+techniques,+2010&lr=&source=gbs_navlinks_s

Sayood, K. (2012). *Introduction to data compression* (4th ed.). Morgan Kaufmann. Retrieved from https://books.google.co.in/books?id=Lhrge2YVpBwC&dq=data+compression+techniques,+2012&lr=&source=gbs_navlinks_s

Storer, J. A., & Szymanski, T. G. (1982). Data compression via textual substitution. *Journal of the ACM (JACM)*, *29*(4), 1982.

Welch, T. A. (1984). A Technique for High-Performance Data Compression. *Computer*, *17*(6), 8–19.

Williams, R. N. (1991). An Extremely Fast ZIV-Lempel Data Compression Algorithm. In *Data Compression Conference*. IEEE.

Winters, K. D., Owsley, P. A., French, C. A., Bode, R. M., & Feeley, P. S. (1996). Adaptive data compression system with systolic string matching logic. United States. Retrieved from http://www.google.co.in/patents/US5532693

Wolff, F. G., & Papachristou, C. (2002). Multiscan-based Test Compression and Hardware Decompression Using LZ77. In *Test Conference, 2002. Proceedings. International*. IEEE.

http://corpus.canterbury.ac.nz/descriptions/

Yang, J., Zhang, Y., & Gupta, R. (2002). Frequent value compression in data caches. In *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000* (pp. 258–265). IEEE. https://doi.org/10.1109/MICRO.2000.898076

Yang, L., Lekatsas, H., & Dick, R. P. (2006). High-performance operating system controlled

memory compression. In *Proceedings of the 43rd annual conference on Design automation - DAC '06* (p. 701). New York, New York, USA: ACM Press. https://doi.org/10.1145/1146909.1147086

Zhang, Y., Yang, J., & Gupta, R. (2000). Frequent value locality and value-centric data cache design. *ACM SIGOPS Operating Systems Review*, *34*(5), 150–159. https://doi.org/10.1145/384264.379235

Ziv, J., & Lempel, A. (1977). A Universal Algorithm for Data Compression. *IEEE Transactions on Information Theory*, *23*(3), 337–343.

Ziv, J., & Lempel, A. (1978). Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory*, *24*(5), 530–536.